Towards Efficient Processing of General-Purpose Joins in Sensor Networks

Mirco Stern, Erik Buchmann, Klemens Böhm
Universität Karlsruhe (TH), Germany
{Mirco.Stern|Buchmann|Boehm}@ipd.uni-karlsruhe.de

***Abstract*— Join processing in wireless sensor networks is difficult: As the tuples can be arbitrarily distributed within the network, matching pairs of tuples is communication intensive and costly in terms of energy. Current solutions only work well with specific placements of the nodes and/or make restrictive assumptions. In this paper, we present SENS-Join, an efficient general-purpose join method for sensor networks. To obtain efficiency, SENS-Join does not ship tuples that do not join, based on a filtering step. Our main contribution is the design of this filtering step which is highly efficient in order not to exhaust the potential savings. We demonstrate the performance of SENS-Join experimentally: The overall energy consumption can be reduced by more than 80%, as compared to the state-of-the-art approach. The per node energy consumption of the most loaded nodes can be reduced by more than an order of magnitude.**

I. Introduction

Wireless sensor networks (WSN) as a measuring technology have important applications ranging from environmental monitoring to industrial maintenance. Such networks consist of a large number of nodes that are equipped with sensors, allowing for observations at a high spatial resolution. The nodes have constrained communication and computation capabilities and are usually battery operated. Therefore, *energy-efficiency* is of utmost importance. Typically, sensing and communication dominate the power consumption by orders of magnitude compared to computation and accessing RAM [1], [2], [3].

Our concern is the performance of join queries in WSNs. The join operator is important to acquire sensor data: It allows to relate measurements taken at *different* nodes. At the same time, the join acts as a filter, allowing any subsequent analysis to concentrate on issues of interest.

Example 1: Think of a climate researcher who is interested in the minimal distance between two points with a temperature difference of more than ten degrees. This query is expressed in SQL using a join:

```
SELECT MIN(distance(A.x, A.y, B.x, B.y))
FROM Sensors A, Sensors B                    (Q1)
WHERE A.temp - B.temp > 10.0
ONCE
```

□

Briefly, Relation `Sensors` serves as a database abstraction of the WSN. Every node is represented by a tuple with one attribute per sensor of that node (e.g., temperature, light). `ONCE` specifies that the query refers to the current state of the network ("snapshot query"), cf. Section III.

Example 2: The researcher is interested in the correlation of humidity and pressure with the temperature. For his analysis, he acquires the humidity (pressure) readings of pairs of nodes which observe a similar temperature (difference < 0.3 degrees). To exclude the influence of spatial correlation, he requires a minimum distance of 100 m:

```
SELECT |A.hum - B.hum|, |A.pres - B.pres|
FROM Sensors A, Sensors B
WHERE |A.temp - B.temp| < 0.3                (Q2)
AND distance(A.x, A.y, B.x, B.y) > 100
ONCE
```

□

These queries will serve as our running examples.

As sensing costs are orthogonal to the join algorithm, minimizing the communication is key for processing joins energy-efficiently [1]. For query operators such as selection and projection, communication can be minimized based on the following key property: A node can decide *locally* that some data is not required for the result and can refrain from sending it [4], [5]. For the join, the picture is different: A node does not know if there exists a join partner for a tuple somewhere in the network. Most notably, tuples which have to be joined can be arbitrarily distributed. Thus, matching two tuples is very communication intensive.

Prior join approaches avoid the problem of matching arbitrarily distributed tuples by restricting the types of queries or the tuple distributions. For instance, REED [1] supports the comparison of sensor data to pre-defined patterns, given as a static, external relation. [6] presents a join method for tracking rare events, i.e., one of the relations must contain a few tuples only. In addition, there are methods that restrict the placement of the tuples involved. For instance, some require the tuples to be located in small regions that are close to each other, e.g., [7], [8], [9] (cf. Section II). Such specific requirements restrict applicability. In particular, no join method we are currently aware of can efficiently process queries in the style of Q1 and Q2. Our focus in turn is on efficient *general-purpose* join methods. We call a join method "general-purpose" if it fulfills the following requirements:

Requirement 1 (Join Conditions)
A general-purpose join method must be able to handle any number and any kind of join conditions and join attributes.

Requirement 2 (Tuple Distribution)
A general-purpose join method must be able to efficiently handle queries with arbitrary placements of the tuples involved.

Our approach complements the existing, specialized ones

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.27

which are geared towards a certain scenario. While their performance is very good when they are applicable, the underlying assumptions are strict and are frequently not met.

Currently, the only general-purpose join method is an approach which we refer to as *external join*: It sends the complete tuples from the input relations to the base station where the result is computed. This approach, albeit simple, is sometimes optimal: If the join selectivity is low, the result is larger than the input data. In this case, sending the result to the base station will be more costly than sending the input tuples and performing an external join – with *any* join method. But otherwise, the external join requires sending many tuples that do not contribute to the result, and efficiency is low. Q2 serves as an illustration. If there are only few tuples with similar temperature values that are more than 100m away from each other, the external join is costly.

In this paper, we propose *SENS-Join*, an energy-efficient general-purpose join method for sensor networks. Our goal is to avoid shipping tuples through the network that do not join. To do so, a well known idea from distributed databases is the (N-way) semi-join [10]: A relation is filtered based on the join-attribute values of the other relations. However, semi-join algorithms as proposed earlier [11] are not applicable here: Relations in sensor networks are highly distributed. Applying a conventional semi-join algorithm requires consolidating the relations at one or a few (2, 3) site(s). In sensor networks, this is prohibitively expensive. Prior attempts to deploy the semi-join idea in sensor networks [8], [9] have not resulted in a general-purpose solution (cf. Section II). So far, it is unclear how to efficiently deploy the filtering idea.

SENS-Join addresses this problem. While SENS-Join consists of a *pre-computation* which identifies the tuples that join and a subsequent *final result computation*, the challenge is designing the pre-computation: Its costs can easily exhaust the potential savings. Our design combines centralized computations at the base station with a distributed in-network filtering. At a high level, the pre-computation consists of two steps: (a) The join-attribute values of both relations are sent to the base station where they are joined in order to create a filter. The filter specifies which values will be in the result. (b) The filter is disseminated in the network. Subsequently, all tuples that match the filter are sent to the base station where the result is computed. SENS-Join can process any kind of join query (e.g., equi-joins, similarity-joins, or theta-joins) over any number of join conditions and input relations.

Our main innovations are the following features, which are key to an efficient pre-computation, despite the need to match each pair of join-attribute values:

Information flow: SENS-Join comprises two mechanisms that keep the volume of data small: "Treecut" sometimes sends complete tuples instead of join-attribute values to avoid their transmission in a later step. "Selective Filter Forwarding" is pruning the filter progressively during its dissemination, depending on the data distribution.

Quadtree representation of pre-computation data: We propose a new compact representation of the join-attribute values sent during the pre-computation. Since in SENS-Join this data is independent of the tuples used for the final result computation, we can reduce their accuracy without sacrificing correctness of the final result. In addition, our mechanism exploits spatially correlated sensor values by encoding join attributes using a spatial index. Compact representations like Bloom Filters cannot be applied here since they do not allow for evaluating arbitrary join conditions.

We extensively evaluate the performance of SENS-Join. Our results indicate that SENS-Join can reduce the overall energy consumption by more than 80% compared to the state-of-the-art. It can reduce the per node energy consumption of the most loaded nodes by more than an order of magnitude.

II. Related Work

The energy consumption of operations like projection, selection, and aggregation (e.g., [4], [12]) are well studied. Efficient implementations exist in data management systems for WSNs such as TinyDB [4] or Cougar [5]. In contrast, existing systems do not support join operations well: Cougar does not feature an in-network implementation of joins. The authors argue that an in-network join can sometimes be beneficial [2]. However, no details are provided. TinyDB allows to join tuples that are located on the same node only. Each node can materialize its sensor data for a specified interval. TinyDB provides a join operation between two materializations or a materialization and the current data. Besides these general query processors, there is some work on join processing in specific scenarios. Currently, no efficient general-purpose join method exists.

Join methods for specific types of queries. The following approaches are tailored to specific scenarios or types of queries. The most general among them is REED [1]. It facilitates the join of a static external relation and sensor data, to detect pre-specified patterns, given as a static relation. The idea is to distribute the static relation such that each node can access it at little cost. In contrast, we focus on joins over sensor relations. Yang et al. [6] investigate the tracking of rare events. Due to the tracking aspect, the tuples to be joined stem from different points in time. Similarly to REED, one of the relations is distributed to serve as a filter. The authors show how to suppress distributing the filter relation if the condition is already contained (in part) in an earlier filter. This approach differs from ours in several key aspects. Yang et al. address scenarios where one of the relations is very small (a few tuples). This is reasonable for the tracking of rare events, but is in contrast to a general-purpose join method. Moreover, their approach is restricted to one join attribute besides the mandatory temporal one. We do not impose such constraints. The approach by Yiu et al. [13] joins tuples from neighboring nodes, i.e., the join condition is $distance(A, B) \leq d$ where d is less than the communication range. SENS-Join in turn allows for arbitrary join conditions and tuple distributions.

Join methods for specific node distributions. Bonfils et al. [7] study long-running join queries in WSN. They reduce the problem to a variant of the task-assignment problem and adaptively relocate the operator. [14] processes the join on the

path between the input data and the query issuer. [15] extends the approach for range queries. Coman et al. [8] present (among other methods) a "mediated join" which computes the result at a central location inside the network. [16] refines this join variant by saying how to handle the data flow in order to meet the memory constraints of the nodes. The applicability of all of these approaches is limited. They are only efficient if the input relations are distributed over two small regions. In addition, these regions need to be close to each other, compared to their distance to the base station. Finally, the selectivity needs to be very high.

Semi-join based filtering. With respect to join processing, the filtering idea has been explored originally in the context of distributed databases [11]: The semi-join filters one of the relations based on the join-attribute values of the other relation. This idea has been extended to filtering all of the relations ("N-way semi-join") [10], which corresponds to the goal of SENS-Join. However, the algorithms are not applicable in WSNs, where the relations are highly distributed. Deploying a conventional algorithm requires consolidating the relations. This is exactly what makes the evaluation expensive.

For WSNs, there are two approaches that apply a filtering. Both are restricted to join small regions and require a high selectivity. Besides the mediated join, Coman et al. present a semi-join approach [8]. The design is close to the semi-join in distributed databases. The join-attribute values of one of the relations is broadcast over the nodes of the other relation. Yu et al. propose to construct a histogram synopsis per relation [9] to determine which tuples join and where to optimally join them. However, [9] does not say how to construct a compact histogram to compute a multi-dimensional join. In addition, this design only yields energy savings if two small regions are joined and if the query is highly selective. The design of SENS-Join is different, for various reasons: Computing optimal join locations requires collecting location information if the coordinates are not part of the join attributes. This is a huge overhead. In addition, the optimal locations also have to be disseminated. We justify our design which can process general-purpose join queries with arbitrarily distributed tuples efficiently in Section IV .

III. Preliminaries

Network architecture. Our work is based on a network architecture consisting of hundreds to thousands of stationary sensor nodes. Each node is equipped with several sensors, a processor, a small RAM (e.g., SunSPOTs contain 512 KB RAM), a wireless radio, and it is battery operated. A powered base station serves as an access point. Each node is aware of the nodes within its wireless range, which form its neighborhood. It communicates with nodes other than its neighbors using multi-hop routing.

Declarative Queries. Declarative queries are an attractive interface for collecting data in sensor networks, as they hide technical details [4], [5]. To facilitate such queries, the network is seen as a *(sensor) relation*. For homogeneous networks there is one relation. One can think of it as a relation with one attribute per sensor (e.g., temperature) of the nodes and one tuple per node. If the network is heterogeneous, groups of nodes form different relations.

Queries can be snapshot queries or continuous queries: A *snapshot query* asks for the current sensor readings ("snapshot"). A typical usage is an interactive exploration. A *continuous query* reports the current sensor readings periodically.

Parallelism. All sensor networks we are aware of are dedicated to a specific task. The number of users tends to be limited, e.g., one or two researchers. Therefore, we assume a small number of parallel queries.

Query Processing. A query is input at the base station. The network then disseminates the query by a simple broadcast flooding. Query results are propagated to the base station along a routing tree, with the base station as the root. A routing tree is maintained in a distributed fashion: Based on a periodic beaconing mechanism, each node maintains a parent that minimizes the hop count to the base station (for details cf. TinyOS, collection-tree protocol [17]).

Terminology. We say that *a node belongs to a sensor relation* R if it contributes a tuple T to R.

Definition 1 (Join-Attribute Tuple)
Let Q be a join query. A join-attribute tuple T' is a tuple that results from the projection of a tuple T on the join attributes of Q, i.e., $T' = \pi_{JoinAttr}(T)$.

As an example, a join-attribute tuple of Q2 contains the X- and Y-coordinates and the temperature.

Problem statement. In general, SENS-Join can handle queries conforming to the following structure:

```
SELECT R1.attrs, ..., Rn.attrs
FROM Relation_1 R1, ..., Relation_n Rn
WHERE preds(R1) AND ... AND preds(Rn)
AND join-exprs(R1.join-attrs, ..., Rn.join-attrs)
{SAMPLE PERIOD x | ONCE}
```

We require two or more sensor relations and join conditions that are arbitrary expressions over the join attributes. That is, we do not restrict the kind or number of join conditions. In the special case of a self-join, the `FROM` clause contains the same relation multiple times. Optionally, the `WHERE`-clauses can narrow down the scope of the query. After executing SENS-Join, the join result is available at the base station.

The semantics is the standard SQL semantics with extensions for temporal aspects of sensor data. In particular, we adopt the non-SQL clauses `SAMPLE PERIOD` or `ONCE` from TinyDB [18]: `ONCE` computes the result based on the current snapshot. Thus, `SELECT * FROM Relation_1 ONCE` returns a single tuple from each node that belongs to `Relation_1`. `SAMPLE PERIOD` yields a continuous monitoring. It defines the time interval between *independent* executions of the query. The user receives a result every x seconds based on the most recent snapshot.

IV. SENS-Join

The goal of SENS-Join is *to avoid shipping tuples through the network that do not join* by discarding these tuples inside

the network. The challenge is designing such a filtering step: Its costs could easily exhaust the potential savings.

A. Overview

To fulfill Requirements 1 and 2 ("general-purpose"), our design combines centralized computations with a distributed filtering. We substantiate this design after describing the algorithm (Section IV-E). To separate concerns, we present SENS-Join presuming a robust operation of the network and address node failures and network-related problems in Section IV-F.

Suppose that the query has already been distributed. Then, at a high level, SENS-Join comprises the following steps:

1. Pre-Computation:

- **a. Join-Attribute-Collection**: Collect the join-attribute tuples of all relations at the base station and join them.
- **b. Filter-Dissemination**: Distribute a filter specifying which tuples join.

2. Final-Result-Computation:

The nodes in question send their complete tuples to the base station where the final result is computed.

Finding out which nodes contribute to the result requires the join-attribute tuple of each node. The base station collects them and joins them (Step 1a). The join-attribute tuples that have a partner form the "*join filter*", i.e., the filter is a set of join-attribute tuples. It lets a node decide if it contributes to the result by checking whether the filter contains its join-attribute tuple. Therefore, we need to disseminate the filter in the network (Step 1b). Finally, the base station collects those tuples that actually join and computes the result.

So far, the only savings of SENS-Join result from not sending non-join attributes. To illustrate, with Query Q2 [Q1], the volume of data shipped to the base station in Step 1a is 60% [33%] of the data that is shipped with the external join. However, due to Steps 1b and 2, the total savings would be less than 40% [66%]. As our main contribution, SENS-Join features several mechanisms that further reduce the data volume significantly: Treecut (Sec. IV-B), Selective Filter Forwarding (Sec. IV-C) and a compact representation of the join-attribute tuples (Sec. V) used in Steps 1a and b.

We now provide a top-down description of SENS-Join. To ease presentation, we describe our approach for two relations in the query, referred to as A and B. We will say how SENS-Join handles more relations whenever this is not obvious.

SENS-Join is a distributed process. Its implementation is event-driven, as explained below. Figure 1 presents SENS-Join from the point of view of a single node.

The node wakes up three times altogether, at the beginning of each step. Since a node knows when its children will send their data for the Join-Attribute-Collection (cf. [18]), it sets the wakeup-time accordingly and goes to sleep (Lines 3, 4). When waking up the first time (Line 6), a node receives the join-attribute tuples of its children (Line 7). It then uses its own sensor readings to generate the Tuple T (Line 8). The procedure returns NULL if the node does not belong to either of the Relations A and B, or if T does not meet

```
1  SENS-Join
2
3  //At the end of the query's dissemination:
4  sleepUntilNextStep(); //wait for beginning of SENS−Join
5
6  Join-Attribute-Collection:
7      ReceivedData = collectMessagesFromChildren();
8      T = constructTupleFromLocalSensorData();
9          //returns T = NULL if (T ∉ A) and (T ∉ B)
10     ForwardJoinAttrValues(ReceivedData, T); //cf. IV-B
11     sleepUntilNextStep();
12
13 Filter-Dissemination:
14     JoinFilter = receiveFromParent();
15     ForwardJoinFilter(JoinFilter); //cf. IV-C
16     sleepUntilNextStep();
17
18 Final-Result-Computation:
19     ReceivedData = collectMessagesFromChildren();
20     ForwardCompleteTuples(ReceivedData, T); //cf. IV-D
```

Fig. 1. SENS-Join, at each node

the selection predicates in the `WHERE`-clauses. Finally, the node forwards the join-attribute tuples received along with its own join-attribute tuple to its parent (Line 10, cf. IV-B). This requires projecting its tuple T onto the join attributes. The projection is part of the forwarding procedure and is not shown in Figure 1. A node then sleeps until the beginning of the Filter-Dissemination step (Line 13). Now the join filter is disseminated in the network along the routing tree. A node simply receives (Line 14) and forwards the filter (Line 15, cf. IV-C). As a final step (Line 18), the complete tuples are forwarded along the routing tree (Line 20, cf. IV-D) to the base station. There the query result is computed.

B. Collecting Join-Attribute Tuples

Which data does a node send in this step? In case of two Relations A and B, the tuple of a node might belong to either A, B, or none of the relations. Thus, a node contributes a join-attribute tuple or nothing. For self-joins it is also possible that the node belongs to both relations. However, it still sends one join-attribute tuple only, consisting of the join-attribute values from both relations. The reason is that the join attributes usually overlap (e.g., they are identical in Q1 and Q2). So we avoid sending attribute values redundantly. In summary, each node contributes at most one tuple.

To assess the design of the Join-Attribute-Collection step, suppose that join-attribute tuples are collected in a straightforward way. Each node, starting at the leaves of the routing tree, collects these tuples from its children and forwards them to its parent along with its own tuple. A leaf node solely sends its join-attribute tuple T'. However, the difference to sending the complete tuple is only a few bytes. For instance, the difference of T - T' in Q2 is two attributes. Assuming that each attribute requires two bytes, this corresponds to sending only four bytes less. The important observation is that the energy savings due to sending T' instead of T are negligible[1].

[1]For instance, removing about 10 bytes from a packet incurs a saving in the order of 5% for SunSPOTs or MicaZ. The reason is the huge overhead due to networking-related issues like channel acquisition, synchronization ([19]).

Depending on the number of children, this remains true at the next level of the tree. The problem with these minor savings near the leaves is that at the same time *we risk the costs of sending an additional packet in the Final-Result-Computation phase if a tuple actually contributes to the result*.

Treecut. We avoid this inefficiency as follows: Starting at the leaves of the tree, we send complete tuples for the pre-computation as long as the volume of data that has to be sent is less than a predefined threshold D_{max}. This applies near the leaves of the tree where the forwarding load is small. We use $D_{max} = 30$ bytes (cf. discussion in Sec. IV-E). If the sum of the data that needs to be sent exceeds D_{max} at some node, this node stores the complete tuples of its subtree and switches to sending join-attribute tuples. In the subsequent steps, the node serves as a proxy for its children, i.e., it handles the Final-Result-Computation without requesting data from the children. Intuitively, Treecut reduces the depth of the tree for the following steps. This improves the efficiency of SENS-Join: We do not have to forward the join filter to those parts of the tree that were "cut off". In addition, if there are tuples that join, we have already forwarded them one or two levels up in the tree. Figure 2 presents the forwarding procedure.

```
1  ForwardJoinAttrValues(Set {S_1, ..., S_n}, Tuple T)
2  //S_i: data received from child i
3
4  Set_Of_Full_Tuples FullTuples = ∅;
5  Join_Attr_Structure JoinAttTuples = ∅;
6  for all S_i ∈ {S_1, ..., S_n}
7      if (S_i is Set_Of_Full_Tuples)
8          FullTuples = Union_Full_Tuples(FullTuples, S_i);
9      else
10         JoinAttTuples = Union_Join_Atts(JoinAttTuples, S_i);
11
12 if (Size({S_1, ..., S_n}) + Size(T) ≤ D_max)
13     && (∀S_i ∈ {S_1, ..., S_n}: S_i is Set_Of_Full_Tuples)
14     //use Treecut: hand over data to parent and go to sleep
15     FullTuples = Insert_Full_Tuples(FullTuples, T);
16     send(FullTuples, parent);
17     //query execution is complete:
18     exitQuery();
19 else
20     store FullTuples; //act as proxy for received complete tuples
21     store JoinAttTuples as "SubtreeJoinAtts";
22     ProxyJoinAttTuples = π_JoinAttr(FullTuples);
23     JoinAttTuples = Union_Join_Atts(JoinAttTuples, ProxyJoinAttTuples);
24     T' = π_JoinAttr(T);
25     JoinAttTuples = Insert_Join_Atts(JoinAttTuples, T');
26     send(JoinAttTuples, parent);
27     //sleep until next step - cf. Figure 1
```

Fig. 2. ForwardJoinAttrValues

Due to Treecut, a node either sends complete tuples or join-attribute tuples. Thus, we have to distinguish between two different data structures for transmission. Complete tuples are forwarded as a multiset (`Set_Of_Full_Tuples`). In contrast, we insert the join-attribute tuples into a more elaborate data structure, discussed in Section V (`Join_Attr_Structure`).

ForwardJoinAttrValues starts by merging the data from the children into a single data structure for complete tuples (Line 8) and join-attribute tuples (Line 10), respectively. Then it is determined whether Treecut applies, depending on the amount of data to send (Lines 12, 13). If so, the node adds its tuple to the data received, sends it to its parent, and is done executing the query (Lines 15 - 18). Otherwise, the node stores the complete tuples from its children to act on behalf of them in the Final-Result-Computation step (Line 20). In addition, it stores the join-attribute tuples of its subtree (Line 21). This is used in the Filter-Dissemination step, and Section IV-C will deal with it. Then the node generates the join-attribute tuples of the complete tuples received as well as its own join-attribute tuple T', adds it to the data received, sends it to its parent, and waits for the Filter-Dissemination (Lines 22 - 27).

Memory capacities. Treecut introduces proxies that store the data of their descendants. The amount of memory needed is limited by D_{max} (30 bytes) multiplied by the number of children of a node. The children are a subset of its communication neighbors. Thus, the number of these neighbors can serve as an upper bound, usually around 6 to 15 [3], [8]. In summary, Treecut requires only a small fraction of the capacities of the node (hundreds of KBs, cf. Sec. III).

C. Disseminating the Join Filter

After the base station has received the join-attribute tuples of both relations, it joins them and creates the join filter. SENS-Join now has to disseminate the filter in the network. A simple way to do so would be to forward it along the routing tree. However, based on the following observation, we can significantly reduce the number of packets: *In the Join-Attribute-Collection step, each node gets to know the join-attribute tuples of its descendants.* If a node keeps this knowledge until the Filter Dissemination, it can decide

1) which part of the join filter is relevant for its subtree (which join-attribute tuples appear in it), and
2) whether it is necessary to forward the join filter at all.

Selective Filter Forwarding. The first item means reducing the size of the join filter while being forwarded to the leaves, i.e., the filter is pruned progressively. The second item refers to the situation where none of the tuples from the filter appears in the subtree of a node. In this case there is no need to forward the filter, i.e., the filter is forwarded exclusively into those regions that contain result tuples. Figure 3 shows the forwarding procedure. Since the filter also is a set of join-attribute tuples, the same data structure is used as for the prior collection step (`Join_Attr_Structure`).

```
1  ForwardJoinFilter(Join_Attr_Structure Filter)
2
3  SubtreeFilter = Intersect_Join_Atts(Filter, SubtreeJoinAtts);
4  if (SubtreeFilter ≠ ∅)
5      //send join-attribute tuples of subtree to children
6      broadcast(SubtreeFilter);
7  else
8      //do nothing - the subtree won't be involved in final step
9  //sleep until next step - cf. Figure 1
```

Fig. 3. ForwardJoinFilter

Recall that nodes have stored the necessary knowledge (SubtreeJoinAtts) during the Join-Attribute-Collection step. ForwardJoinFilter simply intersects the set of join-attribute tuples that appear in the subtree with the join filter (Line 3). This yields the set of join-attribute tuples that contribute to the result and are located in the subtree. The node forwards the result of the intersection if it is not empty. Note that especially if the sensor readings are spatially correlated, complete regions of the tree might not have to forward the join filter.

Memory capacities. Selective Filter Forwarding trades memory for transmission costs. The memory requirements for Selective Filter Forwarding are determined by the specifics of the data structure (which we have not yet discussed). Even without knowing the details, observe that it is possible to bound the memory used: A node keeps the join-attribute tuples of its subtree if their size is less than a predefined limit. We use a limit of 500 bytes. To illustrate, this is only a small fraction of the 512 KB of a SunSPOT. Introducing a limit only has a minor influence on the performance of Selective Filter Forwarding since the amount of data exceeds a few hundred bytes close to the root only. However, the mechanism has its main benefit towards the leaves.

D. Final Result Computation

After the filter has been disseminated, this step collects the complete tuples. Conceptually, this is the simplest step: All it does is forwarding the tuples along the routing tree to the base station. Depending on how many tuples actually contribute to the result, the data volume can be very small. Finally, the base station computes the result.

Note that the complete tuple needs to be stored in the first step (cf. Figure 1, Line 8). It is not possible to re-acquire it from the sensors as the sensor readings could have changed since the Join-Attribute-Collection. As any other join algorithm, SENS-Join reads the sensors exactly once.

E. Design Considerations

Parameter D_{max}. We have argued that, if the absolute amount of data (D_{max}) is already small, it can hardly be reduced. Thus the energy savings would be small. This argument holds if the number of packets is not affected, leading to an important constraint: $D_{max} < MAX\_PACKET\_SIZE$. Beyond that, our choice of $D_{max} = 30$ bytes is justified by our experiments: E.g., if the set of tuples exceeds 50 bytes, SENS-Join already achieves a data reduction of about 25 to 30 bytes. This is due to switching to join-attribute tuples as well as to our compact data structure (`Join_Attr_Structure`).

Join Locations. An important design decision is where the join-attribute tuples are joined and where to compute the final result. We perform both computations at the base station. For the final result, the base station is the optimal join location. This is the result of a theoretical analysis which we have carried out [20]. As a rough intuition, this is due to the filtering: The selectivity of joining the filtered relations is low and the result is larger than the input. Thus, sending the result to the base station is more costly than sending the input tuples. For the pre-computation, we found in-network approaches to be superior to using the base station in specific scenarios only. Thus, our choice is better in most cases.

Discussion. [3] proposes using an index ("semantic routing tree") on static attributes to reduce the costs of forwarding queries inside the network. Selective Filter Forwarding is different. Our mechanism prunes the join filter which is forwarded *progressively*. In addition, we do not build a dedicated index tree. We exploit the knowledge available at a node anyhow. Thus, we employ a temporary structure which comes at no additional costs. Our mechanism is also applicable for attributes whose value changes frequently, in contrast to [3].

F. Design for Error Tolerance

One of the most critical issues for algorithms in sensor networks is coping with changes in the network topology, due to links going down and node failures. SENS-Join builds upon tree-based routing. In particular, the collection-tree protocol (CTP) of TinyOS has undergone a series of optimizations to adapt to changes in the topology. By building upon this mature routing technology, we can exploit its error handling mechanisms: An execution of SENS-Join requires the routing tree to be stable for the duration of a single execution. This is in the order of a few seconds. Beyond a single execution, SENS-Join does not maintain any state. If a link goes down during the execution of a query, we rely upon the tree protocol to re-establish the routing structure. Afterwards, we simply re-execute the query. If data loss needs to be avoided, a more elaborate technique for handling link failures would be required that stores the data during the outage. However, while network disconnection is a problem, it is infrequent given the short execution time of queries and we leave a more elaborate error handling as future work.

V. Compactly Representing Join-Attribute Tuples

We now describe the data structure used to represent join-attribute tuples in the first two steps. Our mechanism roughly halves the costs of the pre-computation.

Mechanisms like Bloom Filters cannot serve as a compact representation in our context since they only allow for evaluating equi-joins. Compression algorithms would also be unsuitable: Firstly, they are not targeted towards small data volumes. This results in bad compression ratios for our problem, see Section VI-B. Next, a compression algorithm would introduce a huge overhead [21]: A node must decompress the data received before adding its own tuple. The data then needs to be re-compressed before forwarding it. Our compact representation avoids this problem by computing the primitives $\texttt{Insert}_{\texttt{Join\_Atts}}$, $\texttt{Union}_{\texttt{Join\_Atts}}$, and $\texttt{Intersect}_{\texttt{Join\_Atts}}$ directly on it.

A. Key Ideas of Compact Representation

Our mechanism pursues two goals: Firstly, we minimize the number of bits required to represent a **single join-attribute tuple**. The second goal is to compactly encode a **set of tuples**.

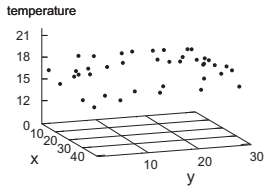

Fig. 4. Distribution of values for 3 join attributes

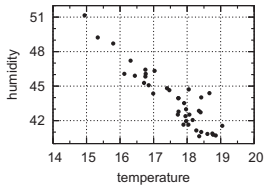

Fig. 5. Distribution of values for 2 join attributes

The key idea towards representing single join-attribute tuples is to perform a quantization of the range of each sensor type. This lets us influence the number of bits. Clearly, a quantization reduces the accuracy of the join-attribute tuples. This is not a problem: As the join-attribute tuples are only used for the pre-computation, we can reduce their accuracy without sacrificing correctness of the final result.

To compactly encode a set of join-attribute tuples we exploit spatial (auto-) correlation of sensor readings. We briefly provide the intuition: Figure 4 shows temperature measurements and their locations, taken from a real-world deployment [22]. In the presence of spatial correlation, sensor readings from nearby nodes are likely to be similar. As a consequence, a set of join-attribute tuples is highly redundant. Our representation eliminates this redundancy by means of a spatial index.

B. Quantization

Conceptually, join-attribute tuples are points in an unbounded, continuous, n-dimensional space. Figure 5 illustrates this perception: Given a query with two join attributes, humidity and temperature, join-attribute tuples are points in a two-dimensional space. The idea of a quantization is to approximate a continuous range of values by a relatively small set of discrete values. Quantization requires us to specify bounds on the ranges ([min, max]) and a resolution (step size) for each dimension. The outcome is a *restricted*, *discrete*, n-dimensional space. To complete the quantization we need to assign a symbol to each multidimensional cell. This symbol encodes a join-attribute tuple that falls into the cell. In other words, we need a numbering which maps a multidimensional point to one dimension, i.e., a space-filling curve.

For the numbering it is important that numbers corresponding to nearby join-attribute tuples are similar in order to keep the spatial correlations. Z-ordering accomplishes this, cf. Figure 6a and b. Besides its good locality-preserving behavior, Z-ordering is easy to compute. This is important in WSNs. We compute the Z-number of a point by bit interleaving of the coordinates of each dimension, see Figure 6c.

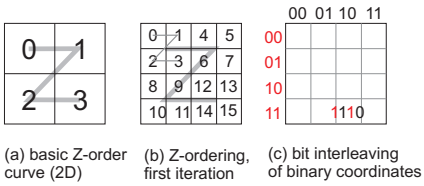

Fig. 6. Z-ordering

We now turn to two important details: Firstly, we need to determine an appropriate range of values ([min, max]) and a resolution of each dimension. This is done at the base station and is disseminated independent of a query. The second aspect refers to computing Z-numbers. The problem is that a sensor measurement might fall outside of the specified range. To ease presentation, we discuss the second aspect first.

Computing Z-numbers. Each node needs to encode its join-attribute tuple which is computing the Z-number. This is presented in Figure 7.

```
1  //compute size of each dimension
2  for all dimensions i
3      SizeOfDim[i] = {(MaxVal[i] − MinVal[i]) · 1/Resolution[i]} + 1;
4      SizeOfDim[i] = roundUpToPowOf2(SizeOfDim[i]);
5      BitPerDim[i] = log(SizeOfDim[i])
6
7  EncodeTuple(Tuple T')
8
9  //compute coordinates (P[i]) of T' in each dimension
10 for all dimensions i
11     P[i] = {(T'[i] − MinVal[i]) · 1/Resolution[i]};
12     if (P[i] < 0)
13         P[i] = 0;
14     if (P[i] ≥ SizeOfDim[i])
15         P[i] = SizeOfDim[i] − 1;
16 //apply bit interleaving to encode P (T')
17 Z = InterleaveBits(P, BitsPerDim);
18 return Z;
```

Fig. 7. EncodeTuple

As a prerequisite for the bit interleaving, we need to know the length of the coordinates. This is implied by the number of cells in each dimension (Lines 2 - 5). E.g., in Figure 6c, we need two bits for each dimension. We compute the number of bits for each dimension separately as, in general, the dimensions are not of equal size. In this case, each dimension contributes to the bit interleaving until its bits are exhausted. `EncodeTuple` starts by computing the coordinates of T' (Lines 10 - 15). By doing so, we ensure that they are within the specified range ([MinVal, MaxVal]) (Lines 12 - 15). This is necessary since the estimated range might be too narrow. In this case we map the value to the boundary of the corresponding dimension. We discuss this solution subsequently. Finally, `InterleaveBits` computes the encoding (Line 17).

Specifying Ranges and Resolution. The following parameters need to be specified: MinVal$[i]$ and MaxVal$[i]$ to bound each dimension as well as their resolution (Resolution$[i]$). These ranges are specific to the environment of the WSN. It is therefore possible to fix them while setting up the network. While elaborate techniques exist for estimating the values, e.g.,

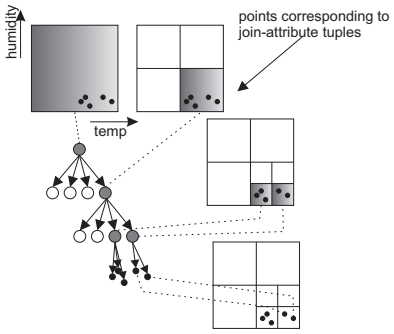

Fig. 8. Construction of the quadtree

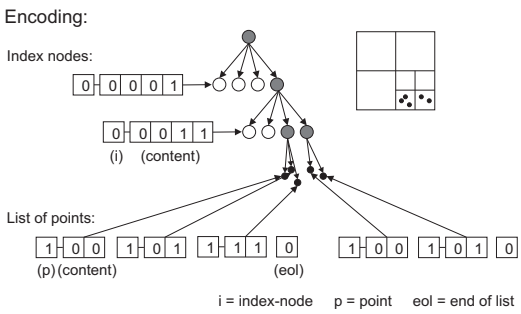

Fig. 9. Encoding of a quadtree

[19], for our purpose reasonably good estimates are sufficient. If our estimated range is too large, we might need more bits to encode a point. But since our domain grows in steps of powers of two (Line 4), a moderate overestimation is not critical. E.g., there is no difference whether we specify a range containing 600 values or 900 values: They both are in the interval $[512, 1024]$ and require 10 bits. In contrast, if the range is too narrow, a value might be outside of it. `EncodeTuple` maps such a point to the boundary of the range. In the worst case this wrongly yields join-attribute tuples that match, and we unnecessarily send their complete tuples. But this affects only a few tuples unless the range is much too narrow.

The idea behind a quantization is to have a coarser resolution to reduce the number of different values per dimension. Again, the resolution has no impact on the correctness of the result. If it is too fine, we need more bits to encode each point. If it is too coarse, complete tuples might be sent erroneously[2]. We found out experimentally that the performance of SENS-Join is insensitive to the resolution used for the precomputation as long as it is not too coarse. Thus, we simply use a fixed resolution for a particular environment. E.g., in our experiments we used steps of 0.1°C for the temperature and of 1m for the X- and Y-coordinates.

C. Representing Points Using a Spatial Index

To encode a *a set of points* our idea is to use a spatial index. A region quadtree [23] is a good choice: Firstly, our goal is to achieve a compact encoding. A region quadtree is based on a regular decomposition of an n-dimensional universe. The 2^n subspaces resulting from a partition are of equal size. This is advantageous for our compact representation since it is not necessary to encode how to divide the space. Secondly, a quadtree is closely related to Z-ordering. The Z-number of a point corresponds to the sequence of quadrants that results from a traversal of a quadtree down to the point.

Encoding sets of points with a quadtree. To illustrate the intuition, assume a query with a single join attribute and the two (discretized) values 23.2°C and 23.4°C. To eliminate redundancy, we could represent them as 23°C plus the relative remainders 2 and 4, respectively. As an abstract illustration, Figure 8 shows a quadtree for five join-attribute tuples from a two-dimensional example. Each tuple corresponds to a point in a two-dimensional, quantized space. Each index node of the quadtree corresponds to a region. At each level of the tree, all dimensions are partitioned into halves. Redundancy within the set of points is eliminated as follows: The index indicates the region which is common to all of the points. If we encode each point relative to the region, then the index node represents what the points have in common. A relative remainder contains information that is unique to a point. Thus, we represent a set of points by the index nodes plus the remainders.

Since quadtrees are well-known, we restrict the discussion of the details to two aspects: Firstly, we need a pointerless representation of the tree. This is because we use it as a wire format. In addition, the use of pointers negatively affects the space requirements of the data structure. The second aspect refers to a general design decision with respect to quadtrees: The decomposition into subspaces is usually continued until the number of points is below a given threshold t [24]. Thus, we need a criterion to decide when to stop the decomposition.

Pointerless representation. There has been a lot of interest in pointerless quadtree representations [23]. We represent the tree as a bitstring consisting of index nodes and the points which are given relative to the path. Figure 9 shows these elements. The pointerless representation is obtained by storing them in the order of a depth-first traversal of the quadtree. This allows us to easily implement $\texttt{Union}_{\texttt{Join\_Atts}}$ and $\texttt{Intersect}_{\texttt{Join\_Atts}}$. We elaborate on this in Section V-D.

The details of our encoding are as follows: An index node starts with a '0' bit specifying that it is an index node. The remaining bits of an index node encode which of the quadrants at the subsequent level is present in the tree. If the number of points in a quadrant is below the threshold, the points are given as a list. A leading '1' indicates a point. Their encoding is relative to the path and contains only the position within the current quadrant. Thus, as the size of the quadrant becomes smaller with every level, so does the relative encoding of a point. As shown in Figure 9, we also need to mark the end of a list of points. This is done by appending a '0'.

Decomposition threshold. In general, the threshold t for stopping the recursive decomposition of a quadtree depends on

[2]As we reduce the resolution, we need to adjust the join of the precomputation not to miss a joining tuple. This is where the false positives come from. These modifications are straightforward for standard Θ-joins.

its use. In Figure 8, t equals 3, so a set of three points is listed explicitly. As our goal is a compact representation, t depends on the number of bits required for a further subdivision vs. the number of bits required for explicitly listing the points. Recursively subdividing a quadrant costs inserting an index node into the tree (cf. the encoding in Figure 9). In contrast, since the points are specified relative to the current path, this subdivision reduces the number of bits of each point. In general, the idea is to compare both solutions and to stop the decomposition if a list of points is shorter.

Encoding of relation membership. To ease presentation, we have omitted so far how to encode which relation a join-attribute tuple belongs to. This is important for the base station to do the join. We prefix each point with two bits ("relation flags") indicating that the point belongs to Relation A ('10'), B ('01'), or to both relations ('11'). As a consequence of prefixing the points, when inserting them into the quadtree, the topmost index node represents the relation flags.

D. Computing Low-Level Primitives

Recall that compression algorithms are not helpful in our context, due to repeated compression/decompression [21]. A strength of our quadtree representation is that $\texttt{Union}_{\texttt{Join\_Atts}}$ and $\texttt{Intersect}_{\texttt{Join\_Atts}}$ can be computed directly on it. There is no need to recover the original tuples. Note that both primitives operate on sets (sets of join-attribute tuples represented as quadtrees) and obey the usual semantics.

$\texttt{Union}_{\texttt{Join\_Atts}}$ is similar to the merge step in Mergesort and can be done in one pass over the data: It merely requires a traversal of the two trees in parallel. Performing these operations directly on the quadtrees is simple due to the depth-first order. Since quadtrees follow a regular decomposition scheme, the shape of the tree is independent of the order of the points being added. We omit the pseudocode for $\texttt{Union}_{\texttt{Join\_Atts}}$ and $\texttt{Intersect}_{\texttt{Join\_Atts}}$ since these operations are well-known. In fact, the ability to perform set operations quickly is one of the reasons for the popularity of quadtrees [23].

VI. Experimental Study

To demonstrate the efficiency of SENS-Join we implemented a prototype in the ns-2 network simulator [25]. We compare its performance to the *external join* which sends both input relations to the base station and joins them. Recall that, despite its simplicity, the external join is optimal if the selectivity is very low. In addition, the external join outperforms the specialized join methods mentioned in Section II in each of our experiments. This is because the latter require very specific scenarios. Thus, in this article, we do not report on comparisons to them.

Our implementation of the external join is state-of-the-art: We aggregate different tuples as they move up the routing tree to reduce the number of packets for the external join. Further, we perform projections and selections as early as possible.

Metric. Typically, sensing and communication dominate the power consumption [1], [2], [3]. As sensing is the same for every join method, we focus on communication.

Choosing a metric for measuring the communication costs is difficult. In general, using the *number of transmissions* (networking packets) is more appropriate than counting bytes transferred, due to the huge per packet overhead. However, this is closely tied to the maximum packet size. For large packets, e.g., 124 bytes, the amount of data per packet can vary significantly, and the number of packets is not directly proportional to the energy consumption either. We use the number of transmissions as our metric with a maximum packet size of 48 bytes. This is commonly used. We discuss the influence of the maximum packet size in Section VI-A.

We compare the communication costs along two lines: *overall communication costs* and *per node communication costs*. The latter is important due to the routing tree. Nodes close to the root are more loaded than leaf nodes. Thus, the per node metric is more critical: When the energy of the nodes near the root is depleted, the network ceases operation.

General setting. For our experiments, we simulate a random distribution of nodes. We set the communication range of each node to 50m and assume links to be bi-directional. This is a common setting in the networking community [2].

For the scope of this presentation we use a fixed distribution of the physical quantities, emulating real sensor data. Varying the data distribution has two effects: (a) The size of the result may change, and (b) the positions of the nodes contributing to the result may change as well. However, we found that changing the positions of nodes only has a minor influence. Further, to vary the fraction of tuples that join, we can also adapt the join conditions. This is much easier to present, and this is what we do.

Parameters. There is a large number of parameters that influence the efficiency of SENS-Join. As discussed below, they can be reduced to the following parameters: (1) fraction of nodes in the result, (2) ratio of $\frac{\text{join attributes}}{\text{attributes overall}}$, (3) number of nodes (size of the network), and (4) packet size.

The idea behind our approach is to send only the tuples that contribute to the result. Thus, the savings depend on the size of this fraction. A second parameter is the ratio of join attributes over the number of attributes in the query. While the external join sends complete tuples, we only send the join-attribute values during the Join-Attribute-Collection step. Thus, the smaller this ratio, the higher the expected savings. Finally, we are interested in the influence of the network size and of the packet size. The first two parameters in particular determine the form of the queries used in the experiments:

```
SELECT A.att_1,..., A.att_n, B.att_1,..., B.att_n
FROM Sensors A, Sensors B
WHERE join-expr(A.join-atts, B.join-atts)
AND ... AND join-expr(A.join-atts, B.join-atts)
ONCE
```

The join conditions are range conditions in the style of Q1 and Q2, used to vary the fraction of tuples in the result. The queries do not contain selection predicates. These would be handled locally and affect the number of nodes concerned. However, our third parameter already controls this number. Beyond that, we found the influence to be negligible and omit

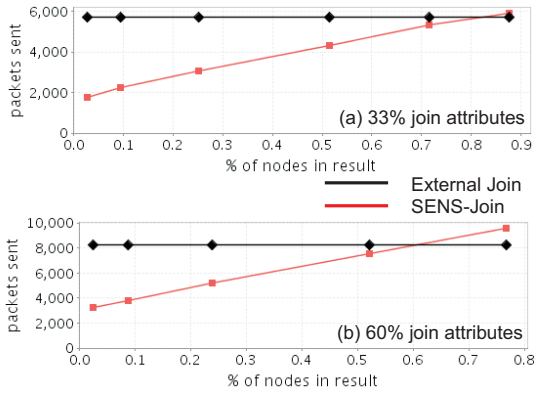

Fig. 10. Overall savings of SENS-Join

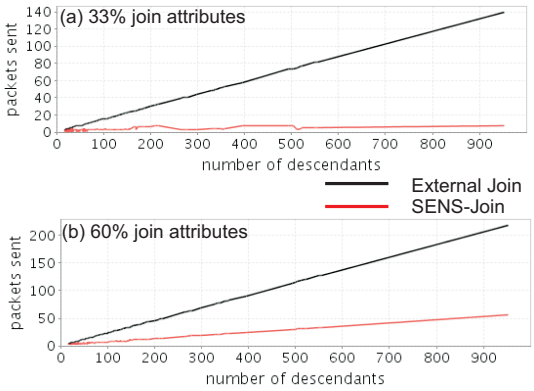

Fig. 11. Per node savings of SENS-Join

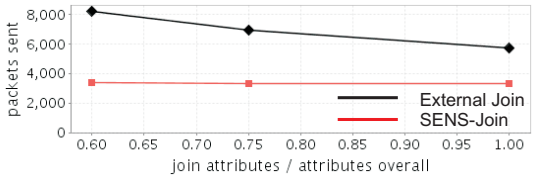

Fig. 12. Influence of the ratio of $\frac{3 \text{ join attributes}}{x \text{ attributes overall}}$

a respective treatment. In addition, we query the same number of attributes from both relations. Otherwise, the tuples sent in the Final-Result-Computation step would be of different size, and the number of transmissions would depend on the fraction of the tuples that are large/small. We found that this parameter has the same effect as varying the number of attributes overall.

Default setting. In each experiment we vary one of the parameters. If a parameter is not varied we use the following default value: The size of the network is 1500 nodes in a 1050m · 1050m area. The fraction of the nodes in the result is 5%. For the ratio of join attributes to attributes overall we will consider two default settings settled towards different ends of the spectrum. The first one is 33% based on one join attribute. The second one is 60% based on three join attributes.

A. *Efficiency of SENS-Join*

Overall communication costs. A first set of experiments determines the overall savings of SENS-Join as compared to the external join. The parameter is the fraction of nodes that contribute to the result. The higher this fraction, the more tuples finally need to be transmitted. Thus, we expect SENS-Join to perform better than the external join unless a high fraction of the tuples joins. There is a break-even point due to the pre-computation of SENS-Join. Figure 10 graphs the results for 33% and 60% join attributes. For the latter, we save up to two-thirds of the overall energy consumption of the external join. For 33% join attributes, the energy savings are up to 80%. They are higher for a smaller ratio of join attributes to attributes overall, as discussed below. Next, SENS-Join is superior until more than 60% (80%) of the nodes join.

Per node communication costs. In the following we investigate the relationship between the number of descendants in the routing tree and the load of the nodes. Due to the forwarding load, nodes with many descendants take up more resources and thus determine the lifetime of the network. Thus, it is critical to reduce the load on these nodes. Figure 11(b) shows that for 60% join attributes the most loaded nodes are unburdened by more than 75%. For 33% join attributes, Figure 11(a) shows a reduction of more than an order of magnitude. This difference increases as the ratio of join attributes to attributes overall decreases. The next paragraph explores this influence in detail.

Ratio $\frac{\text{join attributes}}{\text{attributes overall}}$. We want to verify that a smaller ratio of join attributes to attributes overall increases the savings, and we want to find out if and how these savings are bounded. This also is supposed to justify that our 33% and 60% default settings represent different ends of the spectrum.

The difficulty with the ratio is that the number of combinations is daunting. But many combinations lead to similar ratios (2:4, 4:7, 4:8, etc.) and are close to one of the defaults. Thus, they represent a large number of queries. In contrast, it is possible to increase the ratio to 100%, at least in theory. Though it is difficult to find meaningful queries with very high ratios, the analysis provides a lower bound on the savings.

We now fix the join attributes (join conditions) to have a constant rate of nodes that join (5%). In a first experiment we consider queries with three join attributes. We vary the number of attributes overall from five to three. Clearly, this is the minimum for three join attributes. Figure 12 graphs the results. As expected, the savings increase as the ratio of join attributes to attributes overall decreases. We also see that even for the

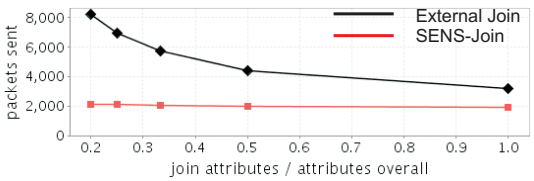

Fig. 13. Influence of the ratio of $\frac{1 \text{ join attribute}}{x \text{ attributes overall}}$

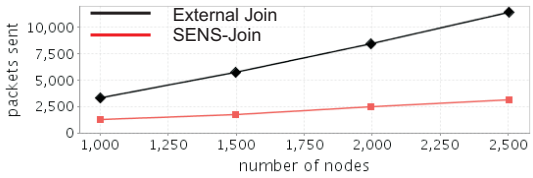

Fig. 14. Influence of the network size

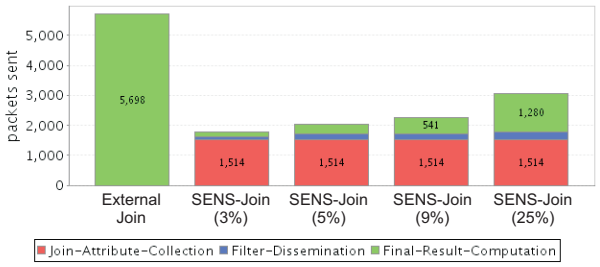

Fig. 15. Costs in the different steps of SENS-Join

worst case of 100% join attributes we save transmissions, compared to the external join. This is because of the quadtree representation. In the second experiment we take one join attribute and vary the number of attributes overall from one to five. The results in Figure 13 confirm our expectations.

Network size. To determine the influence of the network size we vary the number of nodes from 1000 to 2500. At the same time we vary the area of the network to keep the node density constant. We expect the size of the network to have only a small influence on the relative results. If we assume a fixed fraction of tuples that join, the savings are proportional to the size of the network in each step. There is one exception: At the beginning of the Join-Attribute-Collection step when Treecut is applied our method is identical to the external join, and there are no savings. Intuitively, the influence of this initial phase becomes less as the size of the network increases. Figure 14 shows that this expectation holds. The savings are slightly superlinear with the size of the network.

Packet size. The influence of the maximum packet size is difficult to assess. If it is increased, the number of packets is reduced. However, SENS-Join is highly optimized: The data volume of most of the nodes is so small that it already fits into a single 48 byte packet. Therefore, if we increase the packet size, the external join will profit more in terms of packets sent because it sends much more data per packet. However, it is then difficult to draw any conclusions since sending a packet containing, say, 100 bytes is much more costly than sending, say, 30 bytes. The number of transmissions is not directly proportional to the energy consumption any more.

We confirmed our expectations experimentally. Indeed, the external join profits more in terms of the overall number of packets. But for a maximum packet size of 124 bytes, SENS-Join still reduces the number of packets of nodes close to the root by an order of magnitude. This is expected as well: Recall from Figure 11 that SENS-Join reduces the volume of data by much. We omitted the figures due to space constraints.

B. Costs of SENS-Join – Breakdown

We are now interested in an explanation of the savings with SENS-Join. We start by breaking down the number of transmissions to the different steps. In addition, we consider the performance of our compact representation in isolation, and we analyze its influence on the SENS-Join performance.

Costs of the different steps. We now assign the costs of SENS-Join to the different steps. As before, we consider the overall costs if different shares of the tuples join. The costs of the first step (Join-Attribute-Collection) solely depend on the number of join attributes and not on the fraction of tuples in the result. Figure 15 confirms that they are fixed. If there were no result tuples (0% tuples that contribute), there would be no packets in the Filter-Dissemination and Final-Result-Computation steps. Thus, the costs of the first step provide a lower bound on the costs of SENS-Join for a fixed number of join attributes. Let us now look at the costs of the Filter-Dissemination: The number of nodes which need to receive the filter depends on the fraction of tuples in the result. Thus, the smaller this fraction, the more subtrees are pruned.

Performance of quadtree representation. One of the reasons that compression algorithms are unsuitable for our problem is a bad compression ratio for small data volumes. We now compare our compact representation to some well-known compression algorithms of different kinds (cf. [26]): zlib [27] (library form of gzip), which combines LZ77 and Huffman coding, and bzip2, [28], which is based on the Burrows Wheeler Transform. These algorithms do not run on current sensor nodes due to their use of memory and code size. However, by using highly optimized algorithms we provide an upper bound on what can be achieved. We focus on lossless algorithms. Lossy compression leads to incorrect join results.

For this experiment we modified the Join-Attribute-Collection step to either send the raw join-attribute tuples (no compact representation), to use one of the compression algorithms, or to use our quadtree representation. We collect three join attributes: temperature and the location coordinates. This is difficult for our approach: Two of the join attributes are uncorrelated (X- and Y-coordinates). As expected, using a standard compression algorithm results in a poor compression ratio: For 1500 nodes, collecting the join attributes using no compression requires 5619 packets. Bzip2 requires 5666 packets (there is some overhead which increases the volume if it is small) and zlib requires 4571 packets. In contrast, the quadtree representation halves the costs (2762 packets).

Influence of quadtree representation. Finally, we distinguish between the savings due to only sending join attributes and the ones due to the quadtree representation. For this discussion we use the queries from the introduction. Without the quadtree mechanism we save sending two out of five attributes for Q2. Thus, the volume of data is reduced by 40% (66% for Q1). However, the actual savings of the Join-Attribute-Collection step will be slightly smaller. This is because a reduction in the volume of data does not reduce the number of

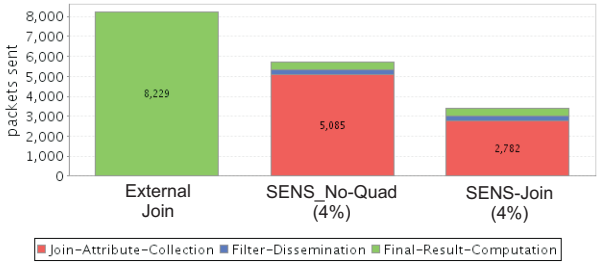

Fig. 16. Influence of quadtree representation

packets if this already is the minimum of a single packet. The numbers in Figure 16 confirm this simple back-of-the-envelope calculation. The Join-Attribute-Collection step needs about 38% less transmissions than the external join. As discussed above, the compact representation halves the volume of data sent. Again, due to the lower bound of a single transmission, some nodes cannot profit from this reduction (Figure 16).

VII. Tradeoffs

SENS-Join can significantly reduce the energy consumption of join processing. However, these benefits are not for free.

Response time. SENS-Join introduces a pre-computation and is thus inferior to the external join regarding response time. In a way, SENS-Join trades response time for energy consumption, as the latter arguably is the most critical resource in WSN. However, the response time of SENS-Join is upper bounded by at most twice the duration of the external join.

Vulnerability to disconnection. If a link happens to go down *during the execution of a query*, our current error handling does not deliver results for the time of the outage (cf. Section IV-F). However, this is an infrequent problem and could be mitigated by a more elaborate error handling.

Memory requirements. Treecut and Selective Filter Forwarding trade memory for transmission costs. As discussed in Section IV, our design accounts for memory restrictions.

VIII. Conclusions and Future Work

General-purpose join queries are difficult to evaluate in WSNs. The tuples are distributed throughout the network, and matching tuples is costly in terms of communication. In this paper, we presented SENS-Join, the first general-purpose join method that can efficiently handle any number of join conditions and arbitrary distributions of the nodes involved. Our design combines centralized computations with a distributed filtering. By thoroughly designing the information flow and by means of a compact representation specific to the pre-computation data, this filtering becomes efficient. SENS-Join is more efficient than the state-of-the-art approach unless a high fraction of the input relations (ca. 60% - 80%) joins. We achieve a reduction of the overall energy consumption by more than 80%. The savings of the most loaded nodes is more than an order of magnitude in some situations. This prolongs the lifetime of the network significantly. As follow-on work we currently investigate if the filtering can be optimized for continuous queries by exploiting temporal correlations.

Acknowledgements. This work was partially supported by the German Research Foundation (DFG) within the Research Training Group GRK 1194 "Self-organizing Sensor-Actuator Networks" (GRK1194). We are grateful to Jörg Sander and M. Tamer Özsu for their comments and to Camillo Scandura for much help.

References

[1] D. J. Abadi, S. R. Madden, and W. Lindner, "REED: Robust, Efficient Filtering and Event Detection in Sensor Networks," in *VLDB*, 2005.
[2] Y. Yao and J. Gehrke, "Query processing for sensor networks," in *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2003.
[3] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The design of an acquisitional query processor for sensor networks," in *SIGMOD*, 2003.
[4] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM TODS*, vol. 30, no. 1, 2005.
[5] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *ACM SIGMOD Record*, vol. 31, no. 3, 2002.
[6] X. Yang, H. B. Lim, M. T. Özsu, and K. L. Tan, "In-network execution of monitoring queries in sensor networks," in *SIGMOD*, 2007.
[7] B. J. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," *Tel. Sys.*, vol. 26, 2004.
[8] A. Coman, M. A. Nascimento, and J. Sander, "On join location in sensor networks," in *MDM*, 2007.
[9] H. Yu, E.-P. Lim, and J. Zhang, "On in-network synopsis join processing for sensor networks," in *MDM*, 2006.
[10] N. Roussopoulos and H. Kang, "A pipeline n-way join algorithm based on the 2-way semijoin program," *IEEE Trans Knowl Data Eng*, vol. 3, no. 4, 1991.
[11] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999.
[12] J. Considine, F. Li, G. Kollios, and J. Byers, "Approximate aggregation techniques for sensor databases," in *ICDE*, 2004.
[13] M. L. Yiu, N. Mamoulis, and S. Bakiras, "Evaluation of spatial pattern queries in sensor networks," University of Hong Kong, Tech. Rep. HKU CS Tech Report TR-2007-02, 2007.
[14] V. Chowdhary and H. Gupta, "Communication-Efficient Implementation of Join in Sensor Networks," in *DASFAA*, 2005.
[15] A. Pandit and H. Gupta, "Communication-Efficient Implementation of Range-Joins in Sensor Networks," in *DASFAA*, 2006.
[16] A. Coman and M. A. Nascimento, "A distributed algorithm for joins in sensor networks," in *SSDBM*, 2007.
[17] R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo, "Tep 123: Collection tree protocol." http://www.tinyos.net/tinyos-2.x/doc/.
[18] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A Tiny AGgregation service for ad-hoc sensor networks," in *OSDI*, 2002.
[19] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong, "Model-driven data acquisition in sensor networks," in *VLDB*, 2004.
[20] M. Stern, E. Buchmann, and K. Böhm, "Where in the sensor network should the join be computed, after all?" in *UKD Workshop*, 2008.
[21] C. M. Sadler and M. Martonosi, "Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks," in *ACM SenSys*, 2006.
[22] http://db.csail.mit.edu/labdata/labdata.html.
[23] H. Samet, "The quadtree and related hierarchical data structures." *ACM Comput. Surv.*, vol. 16, no. 2, 1984.
[24] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comp. Surveys*, vol. 30, no. 2, 1998.
[25] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu, "Advances in network simulation," *Computer*, vol. 33, no. 5, 2000.
[26] K. Sayood, *Introduction to Data Compression*, 2nd ed. Morgan Kaufmann Publishers, 2000.
[27] J. Gailly and M. Adler, "zlib," http://www.zlib.net.
[28] J. Seward, "bzip2," http://www.bzip.org.